PHASE1-1.4 PART 1: Azure Cost Collector - Code Generation

OptiInfra Development Series
Phase: Cost Agent (Week 2-3)
Component: Azure Cost Metrics Collection
Estimated Time: 20 minutes setup + 15 minutes validation
Dependencies: P-01 (Bootstrap), 1.1 (Cost Agent Skeleton), 1.2 (AWS), 1.3 (GCP), 0.10 (Shared Utilities)

📋 Overview

This prompt creates a complete Azure cost collection system that integrates with Azure Cost Management API to gather Virtual Machines, SQL Database, Functions, Storage, and other service costs. It identifies optimization opportunities and stores metrics in ClickHouse for analysis.

🎯 Objectives

By the end of this prompt, you will have:

1. ✅ Azure Cost Management API integration
2. ✅ VM, SQL Database, Functions, Storage cost collectors
3. ✅ Automated cost analysis and anomaly detection
4. ✅ Idle resource identification for Azure
5. ✅ Cost metrics stored in ClickHouse
6. ✅ Comprehensive unit and integration tests
7. ✅ Prometheus metrics for Azure cost collection
8. ✅ Complete multi-cloud comparison (AWS + GCP + Azure)

🔧 Prerequisites

Before starting, ensure:

- ☐ PHASE1-1.2 (AWS) and PHASE1-1.3 (GCP) completed
- ☐ Azure subscription with Cost Management enabled
- ☐ Azure service principal credentials
- ☐ azure-mgmt-costmanagement library installed
- ☐ ClickHouse running with cost metrics tables
- ☐ At least 2 weeks of Azure usage data available

Detailed Windsurf Prompt

Create a complete Azure cost collection system for OptiInfra Cost Agent, following the patterns established in AWS (1.2) and GCP (1.3) collectors.

CONTEXT:
- Multi-agent LLM infrastructure optimization platform
- Cost Agent already has skeleton + AWS + GCP collectors
- Need to collect real Azure cost data via Cost Management API
- Analyze Virtual Machines, SQL Database, Functions, Storage costs
- Identify idle resources and optimization opportunities
- Store metrics in ClickHouse for time-series analysis
- Enable 3-cloud comparison (AWS + GCP + Azure)
- Project root: ~/optiinfra/services/cost-agent

REQUIREMENTS:

1. AZURE COST COLLECTOR BASE (src/collectors/azure/)
   Create base Azure collector class (similar to AWS/GCP base):
   - __init__.py - Module initialization
   - base.py - Base Azure collector with common functionality
     * Azure credential management (service principal)
     * Subscription selection and multi-subscription support
     * Retry logic with exponential backoff
     * Error handling and logging
     * Connection pooling
     * Rate limit handling (Cost Management: 100 calls/min)

   Base class should include:
   - get_credentials() - Load service principal credentials
   - get_client(service_name) - Returns Azure SDK client
   - paginate_results() - Helper for paginated API calls
   - handle_throttling() - Automatic retry with backoff
   - log_api_call() - Track API usage for rate limiting
   - get_all_subscriptions() - List accessible subscriptions

2. COST MANAGEMENT CLIENT (src/collectors/azure/cost_management.py)
   Create Cost Management API wrapper:
   - CostManagementClient class
   - query_costs() - Main cost data retrieval using Usage API
     * Date range: configurable (default: last 30 days)
     * Granularity: Daily, Monthly
     * Dimensions: ServiceName, ResourceLocation, ResourceGroupName

  * Support for filters, aggregations
- get_cost_forecast() - 30-day cost prediction
- get_reserved_instance_recommendations() - RI purchase suggestions
- get_advisor_recommendations() - Azure Advisor cost recommendations
- Data transformation to OptiInfra format

Query structure:
```
{
  "type": "Usage",
  "timeframe": "Custom",
  "timePeriod": {"from": "2025-10-01", "to": "2025-10-31"},
  "dataset": {
    "granularity": "Daily",
    "aggregation": {"totalCost": {"name": "Cost", "function": "Sum"}},
    "grouping": [
      {"type": "Dimension", "name": "ServiceName"},
      {"type": "Dimension", "name": "ResourceLocation"}
    ]
  }
}
```

Response format:
```
{
  "time_period": {"start": "2025-10-01", "end": "2025-10-31"},
  "total_cost": 88000.50,
  "by_service": {
    "Virtual Machines": 58000.00,
    "Azure SQL Database": 16000.00,
    "Azure Functions": 8000.50,
    "Storage": 4500.00
  },
  "by_region": {...},
  "by_resource_group": {...}
}
```

3. VIRTUAL MACHINES COLLECTOR (src/collectors/azure/virtual_machines.py)
Create VM-specific collector:
- VirtualMachinesCostCollector class
- collect_vm_costs() - Per-VM cost breakdown
- identify_idle_vms() - CPU <5%, Network <1GB/day
- identify_underutilized_vms() - CPU <20% for 14+ days

- get_spot_opportunities() - VMs eligible for Azure Spot
- get_reserved_instance_recommendations() - RI purchase suggestions
- analyze_vm_sizes() - Usage patterns by VM size
- get_disk_costs() - Managed disk costs (attached + unattached)
- identify_unattached_disks() - Disks not attached to VMs

Use Azure Monitor for utilization metrics:
- Percentage CPU (14-day average)
- Network In/Out (14-day average)
- Disk Read/Write Operations

Output format:

```
{
  "vm_id": "/subscriptions/.../resourceGroups/.../providers/Microsoft.Compute/virtualMachines/vm-1",
  "vm_name": "vm-1",
  "vm_size": "Standard_D4s_v3",
  "location": "eastus",
  "monthly_cost": 220.00,
  "utilization": {
    "cpu_avg": 14.5,
    "network_gb_day": 280.0
  },
  "optimization": {
    "is_idle": false,
    "is_underutilized": true,
    "spot_eligible": true,
    "rightsizing_recommendation": "Standard_D2s_v3",
    "estimated_savings": 110.00
  }
}
```

4. SQL DATABASE COLLECTOR (src/collectors/azure/sql_database.py)
Create Azure SQL-specific collector:
- SQLDatabaseCostCollector class
- collect_sql_costs() - Per-database costs
- identify_idle_databases() - Connection count = 0
- analyze_storage_costs() - Storage and backup costs
- get_reserved_capacity_recommendations() - Reserved capacity
- identify_elastic_pool_opportunities() - Pool consolidation
- analyze_dtu_usage() - DTU/vCore utilization

Azure Monitor metrics:
- DTU/vCore percentage (14-day average)
- Connection count
- Storage percentage
- Deadlocks

5. AZURE FUNCTIONS COLLECTOR (src/collectors/azure/functions.py)
Create Azure Functions-specific collector:
- AzureFunctionsCostCollector class
- collect_function_costs() - Per-function costs
- analyze_executions() - Execution count and duration
- identify_over_allocated() - Memory > needed
- calculate_optimal_plan() - Consumption vs Premium vs Dedicated
- identify_cold_starts() - Functions with high cold start rate

Azure Monitor metrics:
- Function execution count
- Function execution duration
- Memory working set
- Http response time

6. STORAGE COLLECTOR (src/collectors/azure/storage.py)
Create Azure Storage-specific collector:
- AzureStorageCostCollector class
- collect_storage_costs() - Per-account costs
- analyze_storage_tiers() - Hot vs Cool vs Archive
- identify_lifecycle_opportunities() - Tier transition policies
- calculate_blob_costs() - Blob storage breakdown
- identify_unused_storage() - Empty containers/shares

Cost breakdown:
- Blob storage (by tier)
- File shares
- Queue storage
- Table storage
- Data transfer costs

7. COST ANALYZER (src/analyzers/azure_analyzer.py)
Create comprehensive analyzer:
- AzureCostAnalyzer class
- analyze_all_services() - Aggregate all collectors

- detect_anomalies() - Unusual cost spikes (>20% change)
- calculate_waste() - Total waste across all services
- prioritize_opportunities() - Sort by savings potential
- compare_multi_cloud() - AWS + GCP + Azure comparison
- generate_summary_report() - Executive summary

Analysis output:

```
{
  "total_monthly_cost": 88000.50,
  "total_waste": 38000.00,
  "waste_percentage": 43.2,
  "opportunities": [
    {
      "type": "spot_vm",
      "service": "Virtual Machines",
      "resource_count": 10,
      "estimated_savings": 14500.00,
      "confidence": 0.80,
      "priority": "high"
    },
    {
      "type": "idle_resource",
      "service": "SQL Database",
      "resource_count": 2,
      "estimated_savings": 8000.00,
      "confidence": 0.92,
      "priority": "high"
    }
  ],
  "anomalies": [...]
}
```

8. CLICKHOUSE STORAGE (src/storage/azure_metrics.py)

Create ClickHouse integration:
- AzureMetricsStorage class
- store_cost_metrics() - Daily cost time-series
- store_resource_metrics() - Per-resource metrics
- store_optimization_opportunities() - Discovered opportunities
- query_multi_cloud() - Query AWS + GCP + Azure together

Use existing ClickHouse tables from 0.3:

- cost_metrics (add provider='azure')
- resource_metrics (add provider='azure')
- optimization_opportunities (add provider='azure')

9. API ENDPOINTS (src/api/azure_costs.py)
Create FastAPI endpoints (following AWS/GCP pattern):
- POST /api/v1/azure/collect - Trigger collection
- GET /api/v1/azure/costs - Retrieve cost data
  * Query params: start_date, end_date, service, region, resource_group
- GET /api/v1/azure/opportunities - Get optimization opportunities
  * Query params: min_savings, service, priority
- GET /api/v1/azure/analysis - Get comprehensive analysis
- POST /api/v1/azure/refresh - Force refresh from Azure
- GET /api/v1/multi-cloud/compare - Compare all 3 clouds

All endpoints should:
- Follow same pattern as AWS/GCP
- Validate input with Pydantic models
- Return standardized responses
- Update Prometheus metrics

10. PROMETHEUS METRICS (update src/metrics.py)
Add Azure-specific metrics (mirror AWS/GCP pattern):
- azure_api_calls_total - Counter by service
- azure_api_errors_total - Counter by error type
- azure_cost_collection_duration_seconds - Histogram
- azure_total_monthly_cost_usd - Gauge by service
- azure_waste_identified_usd - Gauge by service
- azure_optimization_opportunities - Gauge by type
- azure_idle_resources_count - Gauge by service
- azure_spot_eligible_count - Gauge
- azure_reserved_instance_coverage - Gauge

11. CONFIGURATION (update src/config.py)
Add Azure settings:
- AZURE_SUBSCRIPTION_ID - Primary subscription
- AZURE_SUBSCRIPTIONS - List of subscriptions (optional)
- AZURE_TENANT_ID - Azure AD tenant
- AZURE_CLIENT_ID - Service principal app ID
- AZURE_CLIENT_SECRET - Service principal password
- AZURE_DEFAULT_LOCATION - Default: eastus

- AZURE_LOCATIONS - List of locations to analyze
- AZURE_COST_LOOKBACK_DAYS - Default: 30
- AZURE_IDLE_CPU_THRESHOLD - Default: 5%
- AZURE_UNDERUTILIZED_CPU_THRESHOLD - Default: 20%
- AZURE_SPOT_SAVINGS_TARGET - Default: 70%
- AZURE_COLLECTION_SCHEDULE - Cron expression

12. TESTING (tests/collectors/test_azure.py)
Create comprehensive tests:
- test_azure_credentials_loading()
- test_cost_management_client()
- test_vm_collector()
- test_sql_database_collector()
- test_functions_collector()
- test_storage_collector()
- test_idle_resource_detection()
- test_spot_opportunity_identification()
- test_cost_analyzer()
- test_clickhouse_storage()
- test_api_endpoints()
- test_multi_cloud_comparison()

Use unittest.mock for Azure API mocking:
- Mock Cost Management responses
- Mock Compute Management list VMs
- Mock Monitor metrics
- Mock SQL Management list databases

13. INTEGRATION TESTS (tests/integration/test_azure_integration.py)
Create E2E tests:
- test_full_collection_workflow()
- test_analysis_pipeline()
- test_storage_retrieval()
- test_api_e2e()
- test_three_cloud_comparison()

14. DOCUMENTATION (docs/azure-collector.md)
Create comprehensive docs:
- Setup instructions
- Service principal creation guide
- Required Azure permissions

- Configuration guide
- API reference
- Example queries
- Troubleshooting guide
- Cost Management API limits

TECHNICAL SPECIFICATIONS:
- Python 3.11+
- azure-mgmt-costmanagement >= 4.0.0
- azure-mgmt-compute >= 30.0.0
- azure-mgmt-monitor >= 6.0.0
- azure-mgmt-sql >= 4.0.0
- azure-identity >= 1.14.0
- Pydantic for data validation
- asyncio for concurrent collection
- tenacity for retry logic
- Rate limiting: 100 requests/minute per subscription
- Batch size: 1000 resources per request
- Error handling: Log and continue
- Caching: Redis for 1-hour cache

BEST PRACTICES:
- Use service principal authentication
- Implement exponential backoff for rate limits
- Cache Cost Management responses
- Parallelize collection across subscriptions
- Use pagination for large result sets
- Validate all Azure API responses
- Log all API calls
- Use Azure Monitor for utilization
- Follow AWS/GCP patterns for consistency
- Store raw + analyzed data separately

ERROR HANDLING:
- Catch azure.core.exceptions.AzureError
- Handle ResourceNotFoundError
- Handle HttpResponseError (rate limit, auth)
- Handle ServiceRequestError
- Log errors but continue
- Retry transient errors (3 attempts)
- Skip resources that fail consistently

SECURITY:
- Never log client secrets
- Use Managed Identity when possible
- Validate all input parameters
- Sanitize resource IDs
- Use Azure Key Vault for secrets
- Rotate credentials regularly
- Limit permissions to minimum required

PERFORMANCE:
- Collect costs in parallel by subscription
- Use batch API calls
- Cache Monitor queries
- Limit date ranges
- Use connection pooling
- Implement request throttling

AZURE-SPECIFIC CONSIDERATIONS:
- Cost Management API requires Enterprise Agreement or Pay-As-You-Go
- Costs update with 24-hour delay
- Azure Spot VMs save 70-90% vs regular
- Reserved Instances require 1 or 3 year commitment
- Azure Hybrid Benefit for Windows/SQL Server
- Azure Monitor metrics have 1-minute granularity
- Multi-subscription requires proper RBAC

FILE STRUCTURE:
services/cost-agent/
```
├── src/
│   ├── collectors/
│   │   └── azure/
│   │       ├── __init__.py
│   │       ├── base.py              # Base Azure collector
│   │       ├── cost_management.py   # Cost Management client
│   │       ├── virtual_machines.py  # VM collector
│   │       ├── sql_database.py      # SQL collector
│   │       ├── functions.py         # Functions collector
│   │       └── storage.py           # Storage collector
│   ├── analyzers/
│   │   └── azure_analyzer.py        # Cost analyzer
```

```
│   ├── storage/
│   │   └── azure_metrics.py          # ClickHouse storage
│   ├── api/
│   │   ├── azure_costs.py            # Azure API endpoints
│   │   └── multi_cloud.py            # 3-cloud comparison
│   ├── models/
│   │   └── azure_models.py           # Pydantic models
│   ├── config.py                     # Updated config
│   └── metrics.py                    # Updated metrics
├── tests/
│   ├── collectors/
│   │   └── test_azure.py
│   ├── analyzers/
│   │   └── test_azure_analyzer.py
│   ├── storage/
│   │   └── test_azure_storage.py
│   ├── api/
│   │   └── test_azure_api.py
│   └── integration/
│       └── test_azure_integration.py
├── docs/
│   ├── azure-collector.md
│   └── azure-service-principal-setup.md
└── requirements.txt                  # Updated with azure-*
```

VALIDATION:
After implementation:
1. Run tests: pytest tests/collectors/test_azure.py -v
2. Test API: curl -X POST http://localhost:8001/api/v1/azure/collect
3. Check metrics: curl http://localhost:8001/metrics | grep azure_
4. Verify ClickHouse: SELECT * FROM cost_metrics WHERE provider = 'azure'
5. Test analysis: curl http://localhost:8001/api/v1/azure/analysis
6. Verify Grafana: Check Azure panels in Cost Agent dashboard
7. Test 3-cloud: curl http://localhost:8001/api/v1/multi-cloud/compare

Generate complete, production-ready code with:
- All collector classes following AWS/GCP patterns
- Complete API endpoints with Pydantic validation
- Comprehensive tests (unit + integration)
- Prometheus metrics integration
- ClickHouse storage layer

- Multi-cloud comparison logic
- Complete documentation
- Service principal setup guide
- Azure RBAC role examples

✅ Success Criteria

After completing this prompt, verify:

1. Azure Connection


bash

```
# Test Azure credentials
python -c "from azure.identity import ClientSecretCredential; print('OK')"
```

2. Cost Collection


bash

```
curl -X POST http://localhost:8001/api/v1/azure/collect
# Expected: {"status": "started"}
```

3. Multi-Cloud Comparison


bash

```
curl http://localhost:8001/api/v1/multi-cloud/compare
# Expected: AWS + GCP + Azure comparison
```

4. Tests


bash

```
pytest tests/collectors/test_azure.py -v
# Expected: All tests pass
```

📁 Files to be Created

- [ ] `src/collectors/azure/__init__.py`
- [ ] `src/collectors/azure/base.py` (~170 lines)
- [ ] `src/collectors/azure/cost_management.py` (~280 lines)
- [ ] `src/collectors/azure/virtual_machines.py` (~360 lines)
- [ ] `src/collectors/azure/sql_database.py` (~220 lines)
- [ ] `src/collectors/azure/functions.py` (~180 lines)
- [ ] `src/collectors/azure/storage.py` (~180 lines)
- [ ] `src/analyzers/azure_analyzer.py` (~300 lines)
- [ ] `src/storage/azure_metrics.py` (~220 lines)
- [ ] `src/api/azure_costs.py` (~280 lines)
- [ ] `src/api/multi_cloud.py` (~200 lines)
- [ ] `src/models/azure_models.py` (~140 lines)
- [ ] `tests/collectors/test_azure.py` (~460 lines)
- [ ] `tests/integration/test_azure_integration.py` (~260 lines)
- [ ] `docs/azure-collector.md` (~100 lines)
- [ ] Updated files (config.py, metrics.py, requirements.txt)

Total: ~3,200 lines of new code

⏱️ Time Breakdown

Task	Estimated Time
Base collector + Cost Management	25 minutes
VM collector	35 minutes
SQL + Functions + Storage	35 minutes
Cost analyzer	25 minutes
ClickHouse storage	15 minutes
API endpoints	20 minutes
Multi-cloud comparison	15 minutes
Metrics integration	10 minutes
Unit tests	40 minutes
Integration tests	25 minutes
Documentation	20 minutes
TOTAL	**~4.5 hours**

Actual time with Windsurf: **20 minutes (code gen) + 15 minutes (validation) = 35 minutes**

🔄 Next Steps

After 1.4 is complete:

NEXT: PROMPT 1.6b - Reserved Instance Workflow

- Complete the 3-cloud cost optimization
- Move to workflow implementation

Document Version: 1.0
Status: 🔄 Ready to Execute
Last Updated: October 21, 2025
Previous: PHASE1-1.3 (GCP Cost Collector)
Next: PHASE1-1.4 PART 2 (Azure Execution & Validation)